CE 6304: Computer Architecture

Group No:17

Project #2

ANALYSIS OF BRANCH PREDICTORS: IMPACT ON DIFFERENT BENCHMARKS

Submitted by

Kalyan Sunder Manivannan - kxm162730

(Contribution:50%)

Jayaramaraja Balaramaraja - jxb162030

(Contribution:50%)

Contents

List of Figures

Introduction:

In this project, we would be discussing about the effect of Branch Prediction on the specific CPU cache configuration and how it improves the performance. Branch Prediction helps to speculate the direction and location of a branch before it's execution. It helps to improve the instruction flow in a pipeline.

Problem Statement:

The goal of this project is to make use of the Branch Predictors available in GEM5 and analyze the Branch Prediction Parameters for a specific CPU cache configuration with different sizes of BTB Entries, Local Predictor, Global Predictor and Choice Predictor for timing based CPU in the GEM5 Simulator. Given with the parameters such as
CPU Model=TimingSimple CPU(timing)
L1icache data size=128kB,
L1dcache instruction size =128KB,
L2 cache size =1MB,
L1Associativity =2
L2 Associativity=4
Block size =64kB
Instructions =500000000
The size of the Branch Target Buffer and Predictors are varied and executed for 6 individual benchmarks, namely 401.bzip2, 429.mcf, 456.hmmer, 458.sjeng, 470.lbm, scimark.

The project is simplified with step by step procedure.

PART 1: ANALYZING THE DIFFERENT BENCHMARKS

All gem5 benchmarks were run on the ce6304 server. Since we ran the simulations for project1 from our machine, we did not face big issues. While preparing the python script to automate gem5 builds, we faced small issues which we could solve by debugging the commands that were issued by the script.

There are three Branch Predictors available in GEM5 Simulator, they are:

1) **Bimode Predictor**: "The bi-mode predictor is a two-level branch predictor that has three seprate history arrays: a taken array, a not-taken array, and a choice array. The taken/not-taken arrays are indexed by a hash of the PC and the global history. The PC only indexes the choice array. Because the taken/not-taken arrays use the same index, they must be the same size.

The bi-mode branch predictor aims to eliminate the destructive aliasing that occurs when two branches of opposite biases share the same global history pattern. By separating the predictors into taken/not-taken arrays, and using the branch's PC to choose between the two, destructive aliasing is reduced. This predictor uses 2-bit saturating counters to predict if a given branch is likely to be taken or not." [10]

2) **Local Predictor**: "A local predictor uses the PC to index into a table of counters. This predictor does not have any branch predictor state that needs to be recorded or updated; the update can be determined solely by the branch being taken or not taken." [11]

3) **Tournament Predictor**: “It has a local predictor, which uses a local history table to index into a table of counters, and a global predictor, which uses a global history to index into a table of counters. A choice predictor chooses between the two. Only the global history register is speculatively updated, the rest are updated upon branches committing or misspeculating.” [12]

Of the three Predictors, Tournament Predictor gives the maximum efficiency in the Branch Prediction as it has a unit which predicts the best Predictor for each Branch. It gives an overall efficiency upto 95%. It would be better if we analyze the performance of the other two Branch Predictors. We would be analyzing on the Bimode Predictor and Local Predictor efficiency with different Benchmarks in the latter sections.

PART 2: CONFIGURING THE BRANCH PREDICTORS IN GEM5

To enable Branch Prediction Simulation in the GEM5 Simulator, changes in the GEM5 files have been made which are explained below.

Config.ini for TournamentBP

[system.cpu.branchPred]
type=TournamentBP
BTBEntries=2048
BTBTagSize=16
RASSize=16
choiceCtrBits=2
choicePredictorSize=4096
eventq_index=0
globalCtrBits=2
globalPredictorSize=4096
indirectHashGHR=true
indirectHashTargets=true
indirectPathLength=3
indirectSets=256
indirectTagSize=16
indirectWays=2
instShiftAmt=2
localCtrBits=2
localHistoryTableSize=2048
localPredictorSize=1024
numThreads=1
useIndirect=false

The GEM5 files have been cloned from the website http://www.gem5.org/Download. The BranchPredictor script in gem5/src/cpu/pred/BranchPredictor.py have been checked whether it contains

*useIndirect = Param.Bool(**False**, "Use indirect branch predictor")*

To add the Branch Predictor of choice to simulate in GEM5, changes to the BaseSimpleCPU.py script in gem5/src/cpu/simple/ BaseSimpleCPU.py have been made. This is the default constructor for the branch predictor which uses the default values from src/cpu/pred/BranchPredictor.py

branchPred = Param.BranchPredictor(BiModeBP(), "Branch Predictor") // for Bimode Branch Predictor

branchPred = Param.BranchPredictor(TournamentBP(), "Branch Predictor") // for Tournament Branch Predictor

branchPred = Param.BranchPredictor(LocalBP(), "Branch Predictor") // for local Branch Predictor

The Branch Predictors are simulated with the Base line configuration for different Benchmarks.

Additional Parameters are added to the stats.txt file to analyze the Branch Predictors. The following files have been modified.

----gem5/src/cpu/pred/bpred_unit.cc (below definition of BTBHitPct)

BTBMissPct

.name(name() + ".BTBMissPct")

.desc("BTB Miss Percentage")

.precision(6);

BTBMissPct = (1 - (BTBHits/BTBLookups)) * 100

----gem5/src/cpu/pred/bpred_unit.hh

Stats::Formula BTBMissPct;

----gem5/src/cpu/simple/base.cc (below definition of numBranchMispred)

t_info.BranchMispredPercent

.name(thread_str + ".BranchMispredPercent")

.desc("Number of branch mispredictions percentage")

.prereq(t_info.BranchMispredPercent);

t_info.BranchMispredPercent=(t_info.numBranchMispred / t_info.numBranches) * 100;

----gem5/src/cpu/simple/exec_context.hh (below

Stats::Formula BranchMispredPercent;

For analysis, extra parameters such as branches_not_stored_in_BTB, unpredictable, predictable_pct, unpredictacle_pct, CPI are calculated through the python script get_data_bm.py and get_data_local.py script.

branches_not_stored_in_BTB = lookups - BTBLookups

unpredictable = Branches - predictedBranches

*predictable_pct = (predictedBranches/Branches)*100*

*unpredictable_pct = (unpredictable/Branches)*100*

*cpi = 1 + ((((l1i_miss_value+l1d_miss_value)*4)+(l2_miss_value*80))/float(inst_cnt)) + ((BranchMispredPercent/100)*15)*

The CPI is calculated by the formula,

$$CPI = 1 + \frac{((L1d_{miss_{inst.}} + L1i_{miss_{inst}})x4) + (L2_{miss_{inst}}x80)}{Instruction\ count} + \frac{BranchMispredPercent}{100}\ x\ BranchMispred\_Penalty$$

Here, **let us assume that BrancMispred_Penalty to be 15 cycles for the X86 CPU**. ie. If a Branch is mispredicted it will take 15 cycles to flush the wrong, speculatively executed instructions related to the branch.

Generating Valid Combinations:

The Baseline configurations are executed ie. For three(3) Branch Predictors and six(6) Benchmarks –(3x6=18 combinations including an extra benchmark : scimark).

Albeit a lot of combinations are possible, we have chosen two Branch Predictors – Local and Bimode Predictor and simulated it for all the six benchmarks by changing the sizes of the BTB Entries, Local predictor, Global Predictor and Choice Predictor.

These conditions generate the valid combinations for the cache configuration

Script in python to generate valid combinations for each benchmark:

(********branch parameters Configuration initialized*******)

```
BTBEntriesValue=['16','32','64','128','256','512','1024','2048','8192','32768']
localPredictorValue=['16','32','64','128','256','512','1024','2048','8192','32768']     #for local Branch Predictor
globalPredictorValue=['16','32','64','128','256','512','1024','2048','8192','32768']   #for Bimode Branch predictor
choicePredictorValue=['4096']
```

#Filter loop starts

```
for bp_var in BP_name:
        for BTB_var in BTBEntriesValue:
                for local_var in localPredictorValue:
                        for global_var in globalPredictorValue:
                                for choice_var in choicePredictorValue:
                                        update_gem5_src(BTBEntriesval = BTB_var,localPredictorval = local_var,globalPredictorval = global_var,choicePredictorval = choice_var, which_BP = bp_var,BPfile = bp_setup_file,BP_paramfile = bp_param_file,gem5_build_dir = gem5_dir,update_BP=1,update_BP_params=1)
                                        processes = []
                                        for index in range(len(test_name)):
                                                for number in range(len(l1dsize_set)):
                                                        final_out_dir = str(output_dir[index])+'/'+str(bp_var)+'_'+str(test_true_name[index])+'_'+str(BTB_var)+'BTBEntries_'+str(local_var)+'local'
                                                        processes.append(subprocess.Popen([gem5_exec,'-d',final_out_dir,gem5_se,'-c',str(test[index]),'-o',str(test_arg[index]),'-I',inst_cnt,'--cpu-type='+cpu_type,'--caches','--l2cache','--l1d_size='+str(l1dsize_set[number])+'kB','--l1i_size='+str(l1isize_set[number])+'kB','--l2_size='+str(l2size_set[number])+'kB','--l1d_assoc='+str(l1assoc_set[number]),'--l1i_assoc='+str(l1assoc_set[number]),'--l2_assoc='+str(l2assoc_set[number]),'--cacheline_size='+str(linesize_set[number])]))
                                        for p in processes:
                                                p.wait()
                        ( ******** Valid combinations generated & corresponding simulations run ******** )
```

About 1000 **combinations** are generated from the above python script.

But, we have selected some realistic values given the time constraints to run the simulations which are discussed below.

Configuration Analyzed under each Benchmark:

Due to time constraint, only limited benchmarks are analyzed. Considering the worst case and best possible configuration, A list of few combinations have been submitted for each benchmark.

Branch Predictor/Benchmark	BiMode	Local
Bzip2	75	100
Mcf	75	100
Lbm	75	100
Hmmer	75	100
Sjeng	75	100
Scimark	75	100
Total	**450**	**600**

So, Total of **1050** combinations are analyzed.

PART 3: BASELINE CONFIGURATION:

The simulations are executed with the Baseline configuration for different Benchmarks. The parameters in the Baseline Configuration are

BTBEntries Size=2048

Local Predictor Size=1024

Global Predictor Size=4096

Choice Predictor Size=4096

The Branch Target Buffer(BTB) entries size should be large enough to hold as much previous target addresses. The local, Global, Choice Predictor size has effect on the prediction of the Branches. The different Benchmarks are simulated with the Baseline Configuration.

SPEC CPU2006 Benchmark

bzip2 (Compression):

401.bzip2 is based on Julian Seward's bzip2 version 1.0.3. This is an integer benchmark. All compression and decompression happens entirely in memory. This is to help isolate the work done to only the CPU and memory subsystem.

401.bzip2's reference workload has six components: two small JPEG images, a program binary, some program source code in a tar file, an HTML file, and a "combined" file, which is representative of an archive that contains both highly compressible and not very compressible files.

Each input set is compressed and decompressed at three different blocking factors ("compression levels"), with the end result of the process being compared to the original data after each decompression step. [5][8]

From the stats.txt created from each configuration, the BTBHitPct, BTBMissPct, BranchMispredPercent and accuracy (total predicted ~ numPredicted/total_branches) is calculated.

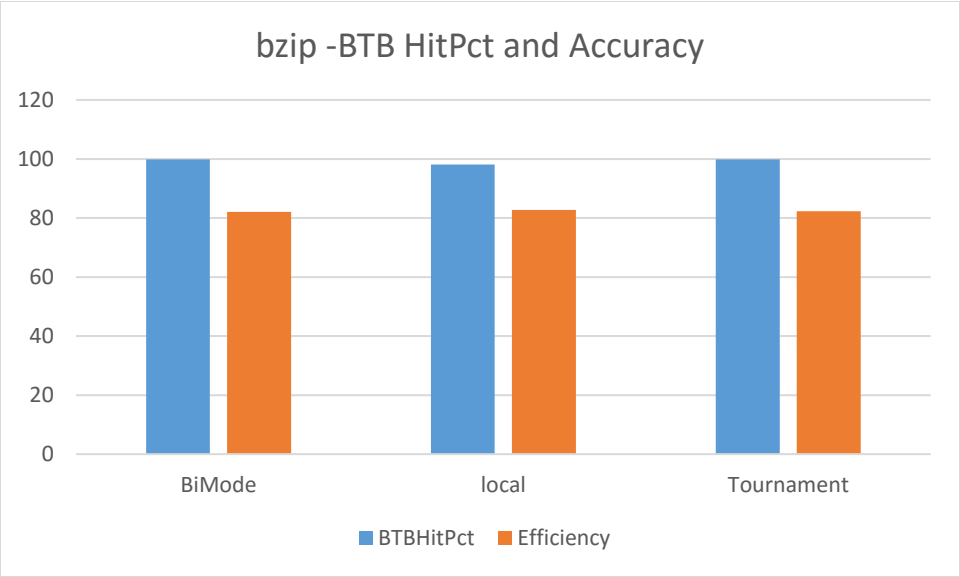


Figure 1: Graph of bzip Benckmark – BTBHit Pct and Overall Accuracy

From the graph, we can see that BTBHitpct is close to cent percent. The BTBHit Pct is good as number of branches are within the size of BTB of 2048. The number of predictions from the total branches in the program is represented as efficiency. i.e. efficiency = number of predicted branches from all available branches. The efficiency is around 80% , which implies that 20% is unpredictable, this indicated that 20% branches are random and cannot be predicted correctly even once by the branch predictor. This can be due to presence of switch statements in the program which branches to random targets.

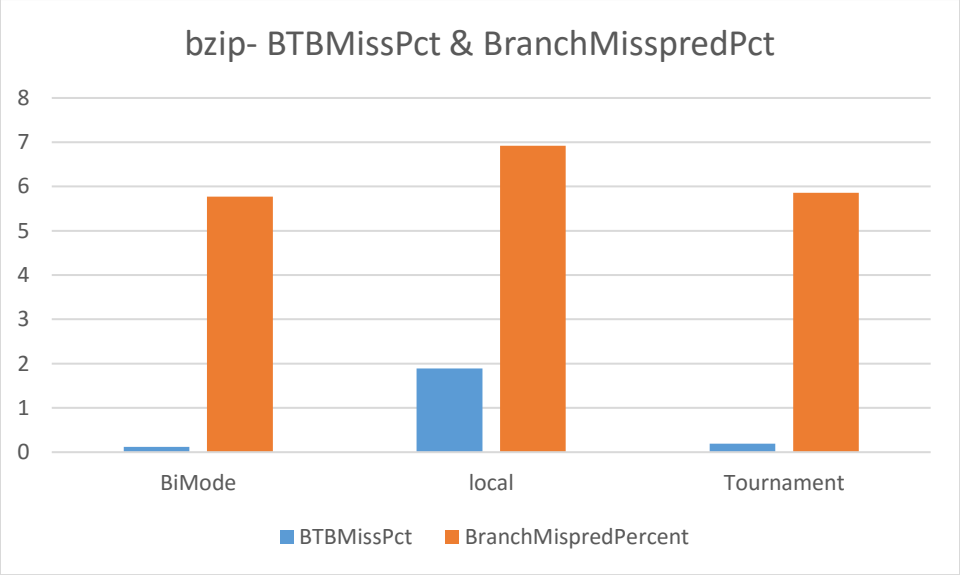


Figure 2: Graph bzip Benckmark – BTBMiss Pct and Branch MisPredict Percent

The data extracted from the stats.txt are given below

bzip	BiMode	local	Tournament
BTBHitPct	99.87691	98.107894	99.804709
BTBMissPct	0.123093	1.892106	0.195291
BranchMispredPercent	5.768177	6.920399	5.860535
Branches	38082430	38082430	38082430
Predicted branches	31273823	31514193	31319965
Efficiency	82.1214	82.752579	82.2425591

The BranchMissPredict Percent is a factor that denotes how much branches which are predictable are wrongly predicted. The unpredictable branch counts are calculated from the total lookups as follows.

branches_not_stored_in_BTB = lookups - BTBLookups

unpredictable = Branches - predictedBranches

*predictable_pct = (predictedBranches/Branches)*100*

*unpredictable_pct = (unpredictable/Branches)*100*

These parameters give the unpredictable percent and the # of Branch targets not stored in BTB. Branch targets not stored in BTB could also mean, for X number of branches, the target is the same , where these X branches are indexing to the same location in the BTB out of the total BTB entries.

mcf (Combinatorial optimization)

This falls under the general category of Combinatorial optimization / Single-depot vehicle scheduling 429.mcf is a benchmark which is derived from MCF, a program used for single-depot vehicle scheduling in public mass transportation. The benchmark version uses almost exclusively integer arithmetic. [5][8]
From the stats.txt created from each configuration the BTBHitPct, BTBMissPct, BranchMispredPercent and accuracy is calculated.

The CPI is calculated from the following formula.

$$CPI = 1 + \frac{((L1d_{miss_{inst.}} + L1i_{miss_{inst}})x4) + (L2_{miss_{inst}}x80)}{Instruction\ count} + \frac{BranchMispredPercent}{100} x\ BranchMispred\_Penalty$$

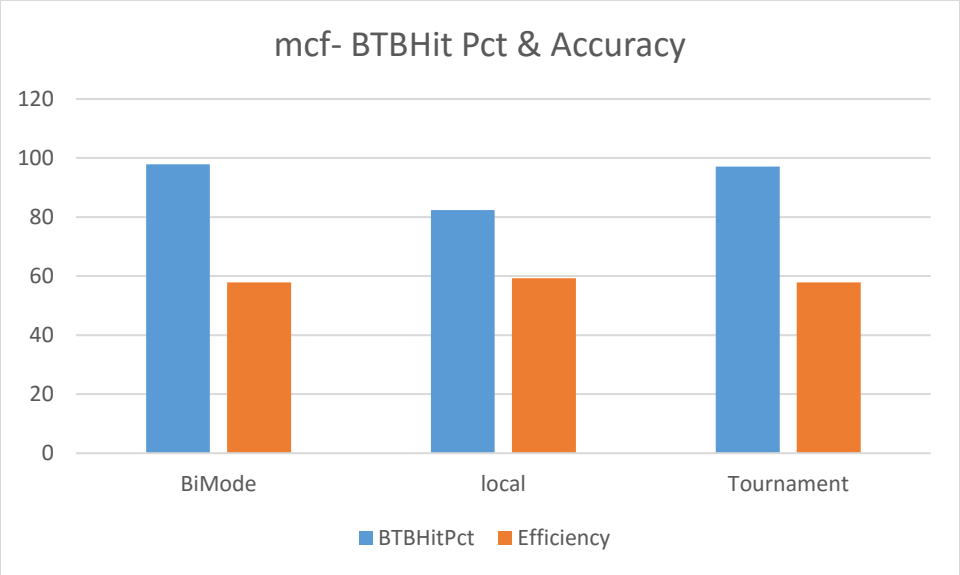


Figure 3: Graph of mcf Benckmark – BTBHit Pct and Overall Accuracy

From the graph, we can see BTBHit Pct and Accuracy are less than bzip because it has more branches than bzip.

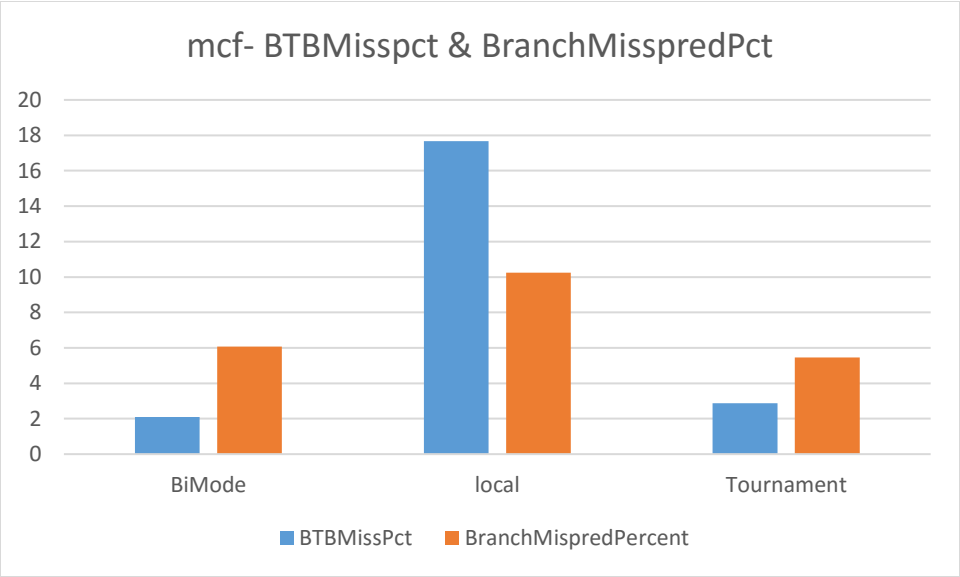


Figure 4: Graph mcf Benckmark – BTBMiss Pct and Branch MisPredict Percent

Clearly, the local predictor is not able to compete due to the higher randomness in the branches from this benchmark.

The data extracted from the stats.txt are given below

mcf	BiMode	local	Tournament
BTBHitPct	97.91033	82.33586	97.12011
BTBMissPct	2.089671	17.66414	2.879893
BranchMispredPercent	6.064948	10.24781	5.449035
Branches	97499524	97499524	97499524
Predicted branches	56380162	57781855	56416046
Efficiency	57.82609	59.26373	57.86289

hmmer (Database Search)

This benchmark uses profile Hidden Markov Models (profile HMMs) or statistical models of multiple sequence alignments, which are used in computational biology to search for patterns in DNA sequences. From the stats.txt created from each configuration the BTBHitPct, BTBMissPct, BranchMispredPercent and accuracy is calculated. [5][8]

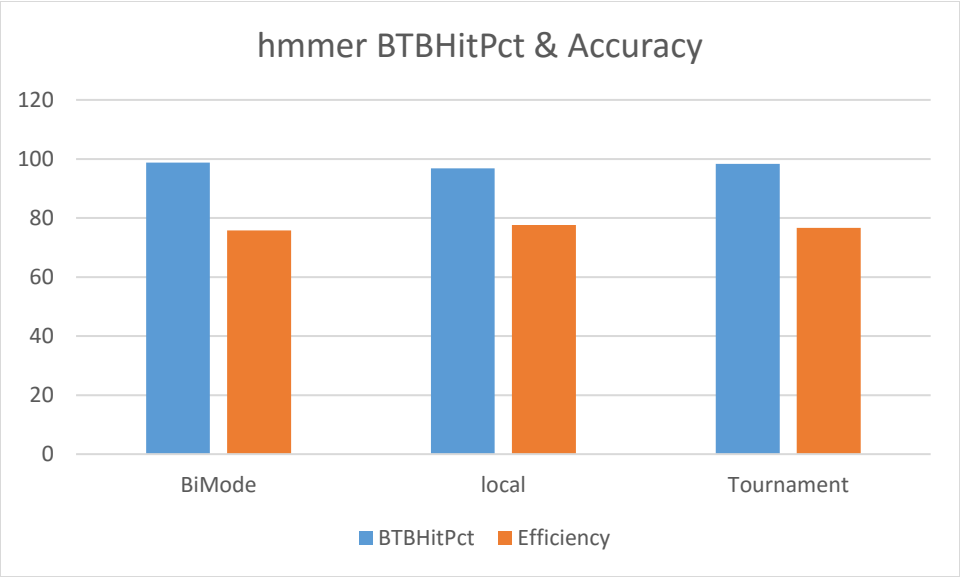


Figure 5: Graph of hmmer Benckmark – BTBHit Pct and Overall Accuracy

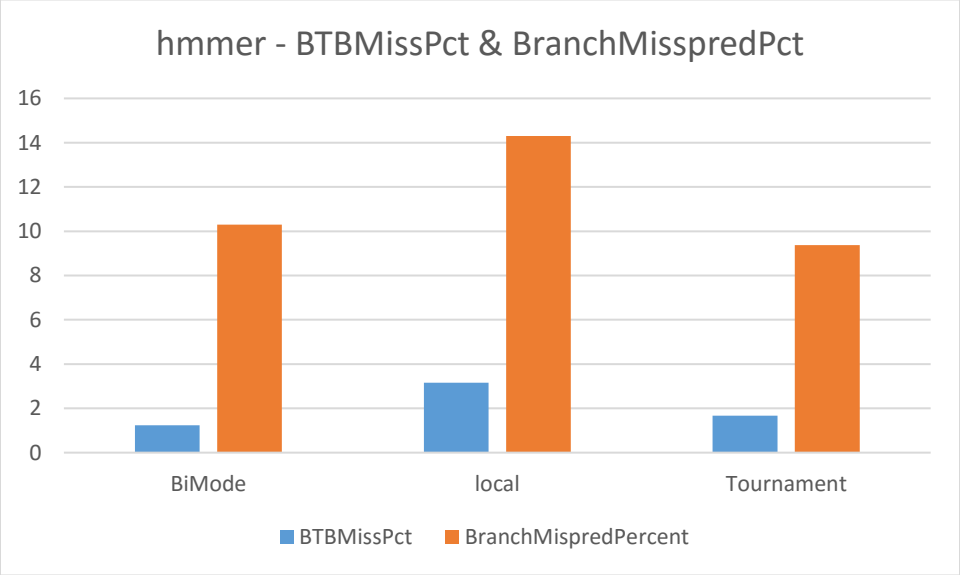


Figure 6: Graph hmmer Benckmark – BTBMiss Pct and Branch MisPredict Percent

The BTBMissPct and BranchMisspredPct are plotted in the graph. Here also, local predictor performs poorly compared to the other predictors.

The data extracted from the stats.txt are given below

hmmer	BiMode	local	Tournament
BTBHitPct	98.76556	96.84549	98.32981
BTBMissPct	1.234444	96.84549	1.670188
BranchMispredPercent	10.29582	14.30742	9.37602
Branches	27643935	27643935	27643935
Predicted branches	20934059	21448780	21192274
Efficiency	75.72749	77.58946	76.66157

sjeng (Artificial Intelligence)

sjeng is based on Sjeng 11.2, which is a program that plays chess and several chess variants, such as drop-chess (similar to Shogi), and 'losing' chess. It attempts to find the best move via a combination of alpha-beta or priority proof number tree searches, advanced move ordering, positional evaluation and heuristic forward pruning. Practically, it will explore the tree of variations resulting from a given position to a given base depth, extending interesting variations but discarding doubtful or irrelevant ones. From this tree the optimal line of play for both players ("principle variation") is determined, as well as a score reflecting the balance of power between the two.

458.sjeng's input consists of a text file containing alternations of

1. a chess position in the standard Forsyth-Edwards Notation (FEN)
2. the depth to which this position should be analyzed, in half-moves (ply depth)

The SPEC reference input consists of 9 positions belonging to various phases of the game. [5][8]

From the stats.txt created from each configuration the BTBHitPct, BTBMissPct, BranchMispredPercent and accuracy is calculated

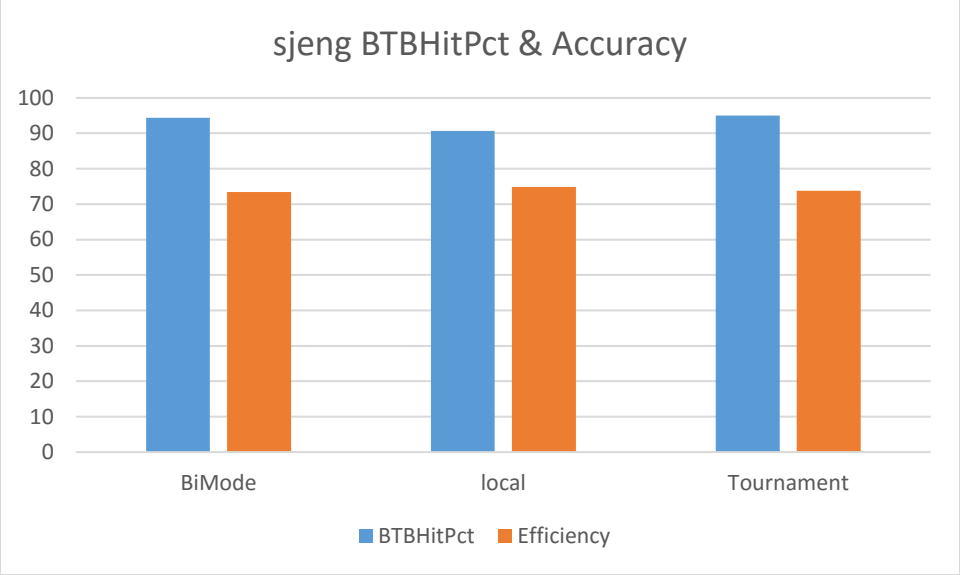


Figure 7: Graph of sjeng Benckmark – BTBHit Pct and Overall Accuracy

The BTBMissPct and BranchMispredPct are plotted in the graph

The data extracted from the stats.txt are given below

sjeng	BiMode	local	Tournament
BTBHitPct	94.33862	90.679889	95.04459
BTBMissPct	5.661385	9.320111	4.95541
BranchMispredPercent	9.485713	14.465739	9.607802
Branches	69585502	69585502	69585502
Predicted branches	51076202	52124181	51355558
Efficiency	73.40064	74.906668	73.80209458

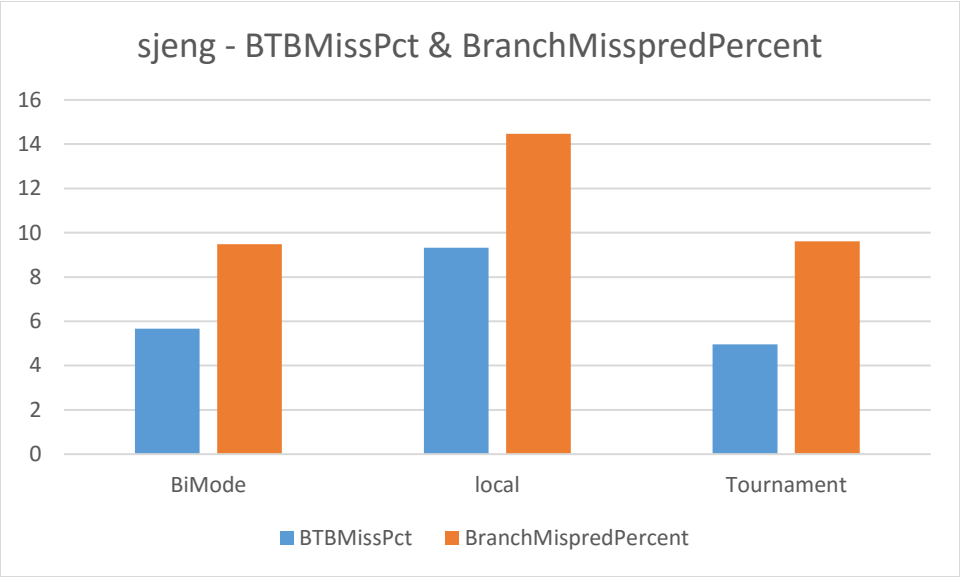


Figure 8: Graph sjeng Benckmark – BTBMiss Pct and Branch MisPredict Percent

FLOATING POINT BENCHMARKS

lbm (Computation Analysis)

This belong to the general category of Computational Fluid Dynmaics, Lattice Boltzmann Method. This program implements the so-called "Lattice Boltzmann Method" (LBM) to simulate incompressible fluids in 3D. For benchmarking purposes, where the SPEC tools are used to validate the solution, the computed results are only stored.

In the Lattice Boltzmann Method, a steady state solution is achieved by running a sufficient number of model time steps. For the reference workload, 3000 time steps are computed. For the test and training workloads, a far smaller number of time steps are computed.

The geometry used in the training workload is different from the geometry used in the reference benchmark workload. Also, the reference workload uses a shear flow boundary condition, whereas the training workload does not. Nevertheless, the computational steps stressed by the training workload are the same as those stressed in the reference run. [5][8]

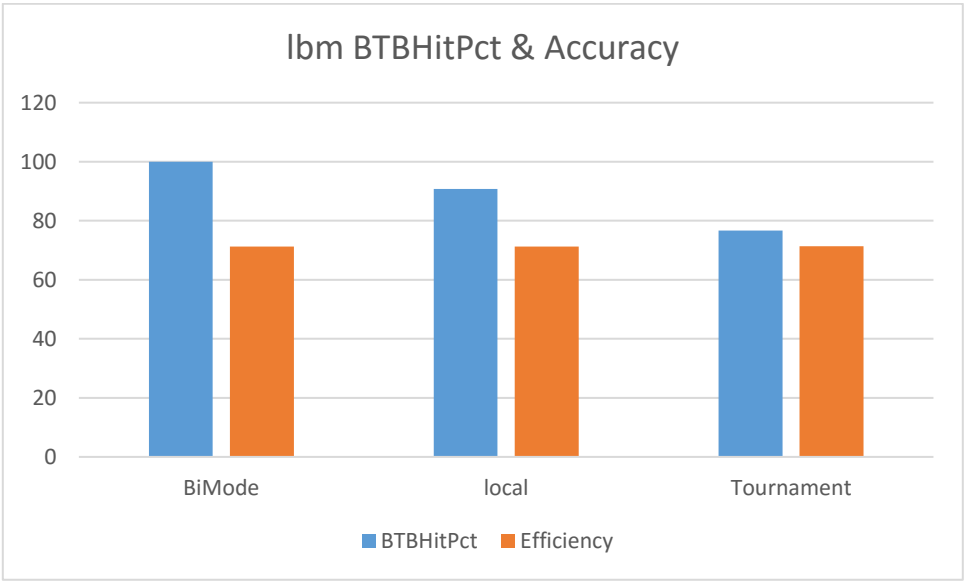


Figure 9: Graph of lbm Benckmark – BTBHit Pct and Overall Accuracy

The BTBMissPct and BranchMispredPct are plotted in the graph

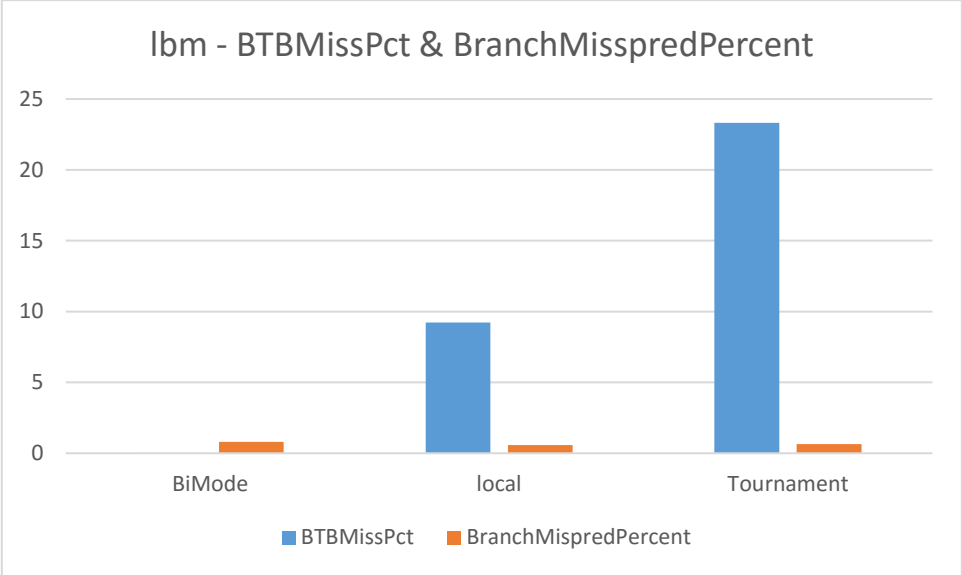


Figure 10: Graph lbm Benckmark – BTBMiss Pct and Branch Miss Predict Percent

Since it is floating point benchmark, less branches are encountered with more computational units. Bimode Predictor predicts Branches better than the other Predictors.

The data extracted from the stats.txt are given below

lbm	BiMode	local	Tournament
BTBHitPct	99.99797	90.7821	76.68112
BTBMissPct	0.002028	9.217898	23.31888
BranchMispredPercent	0.799399	0.586347	0.638224
Branches	18439603	18439603	18439603
Predicted branches	13141285	13142571	13165110
Efficiency	71.26664	71.27361	71.39584

Scimark

In addition to the general Benchmarks given as per specification, we have included extra Benchmark in order to get clean insight of the variation in the branch predictor configuration. This is a floating point benchmark which uses performs FFT, dense Lu matrix factorization. This test runs the Java version of SciMark 2.0, which is a benchmark for scientific and numerical computing developed by programmers at the National Institute of Standards and Technology. This benchmark is made up of Fast Fourier Transform, Jacobi Successive Over-relaxation, Monte Carlo, Sparse Matrix Multiply, and dense LU matrix factorization benchmarks.[5][8]

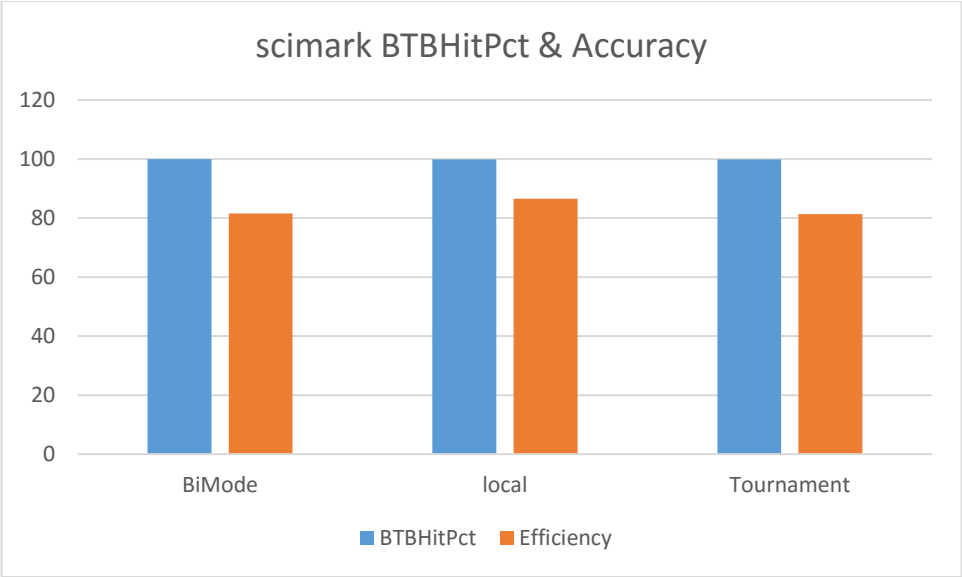


Figure 11: Graph of scimark Benckmark – BTBHit Pct and Overall Accuracy

The BTBMissPct and BranchMispredPct are plotted in the graph

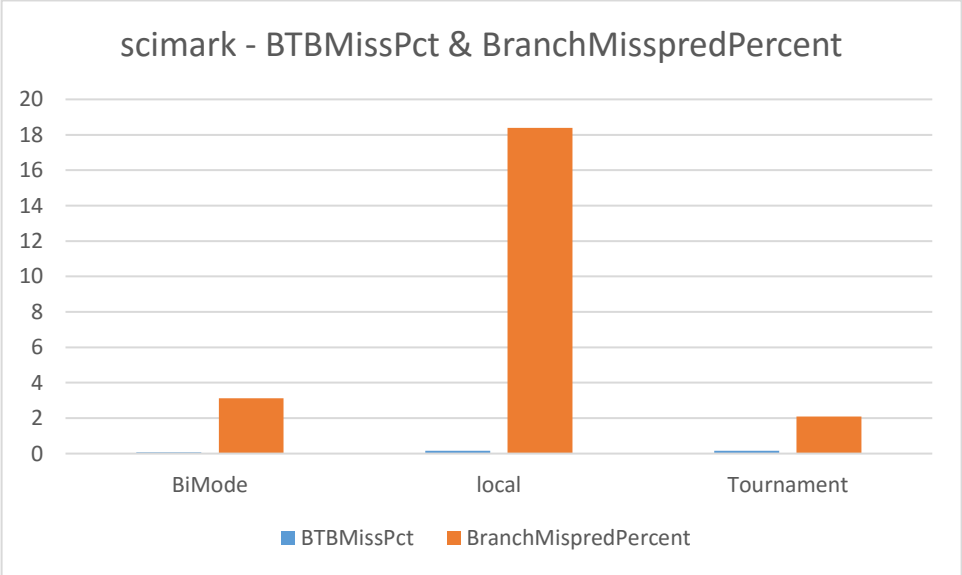


Figure 12: Graph of scimark Benckmark – BTBHit Pct and Overall Accuracy

Here, the tournament predictor is the best among the three. The data extracted from the stats.txt are given below

scimark	BiMode	local	Tournament
BTBHitPct	99.92589	99.8404	99.8477
BTBMissPct	0.074109	0.159598	0.1523
BranchMispredPercent	3.120691	18.39131	2.084505
Branches	11572148	11572148	11572148
Predicted branches	9433099	10006001	9404387
Efficiency	81.51554	86.46624	81.26743

PART 3: ANALYZING PARAMETERS FOR EACH BRANCH PREDICTOR

In this part of the project, we will analyze the effect of BTBEntries, Local Predictor size, Global Predictor size, Choice Predictor size. The best possible configuration for each benchmark is analyzed below.

EFFECT DUE TO BTB ENTRY SIZE:

The Branch Target Buffer is a true cache. Predicting correctly where the branches would go will improve the performance as the CPU can execute the instructions speculatively using the predicted target, this is why the BTB is used. BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches. The BTB entry size directly influence the performance of the predictor. The graph shows the effect due BTBentry size on different Benchmarks.

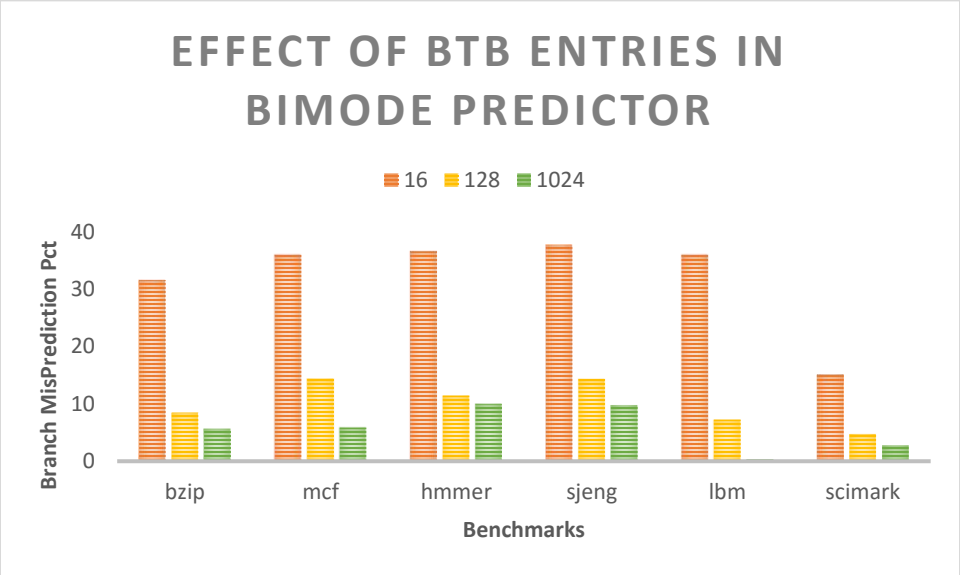


Figure 13: Graph on the Effect of BTB Entries in Branch Predictors

From the graph, it can be inferred that for 16 BTB Entry size the Branch Misprediction Percent is high. For higher size such as 1024 entries, the Branches are predicted better.

The lbm is a floating point benchmark and involves only less branching, so for 1024 BTB entry, there is almost negligible mispredictions.

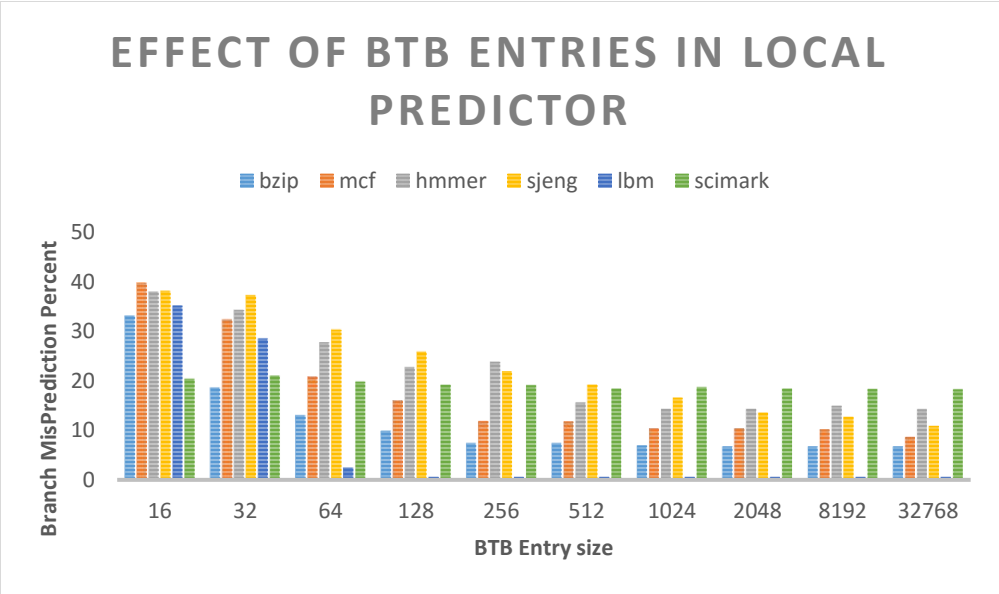


Figure 14: Graph on Effect of BTB Entries in Local predictor

It can be seen that the mispredictions are reduced for higher value of BTB entries. Also, for lbm, misprediction reduces dramatically when BTB entry size increases from 64 entries

EFFECT DUE TO LOCAL PREDICTOR SIZE:

In the Local Predictor, the size of the local Predictor plays an important role in the prediction. It stores local branch history. Here the BTB Entry size is kept constant to 1024 entries and the local Predictor size is varied and its effect on the BranchPredictor parameters are plotted.

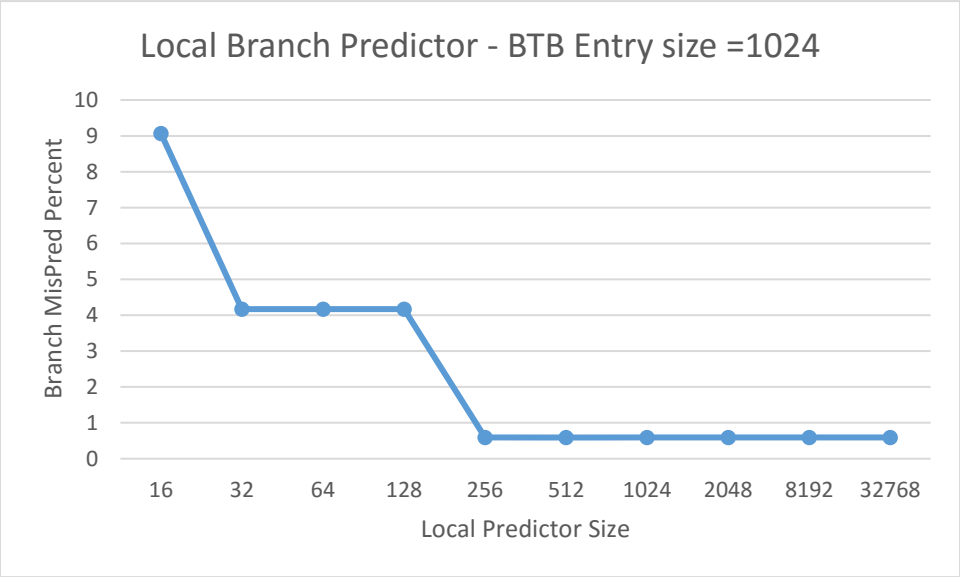


Figure 15: Graph on effect on local Predictor size

ANALYSIS:

As the BTB Entry size is set to 1024, the local Predictor size should be sufficiently large enough to map to the values. If it is too small like 16, then it might have aliasing effect which might lead to mispredicts. For a size of 256 in the local Predictor, the aliasing effect is reduced. MisPrediction remains almost equal for all values greater than 256. Hence optimum value for the 1024 BTB entry size is to have a minimum of 256

of local Predictor size for the particular scenario. Hence, local Predictor size should be considerably large enough to map without aliasing.

EFFECT DUE TO CHOICE PREDICTOR SIZE:

In the Bimode Predictor, the size of the choice Predictors plays crucial role in the prediction. Here the BTB Entry size is kept constant to 1024 and the 16 global Predictor size is kept constant and the size of the choice predictor is varied and its effect on the BranchPredictor parameters are plotted. The condition of higher BTB Entry size and small global predictor size is chosen to analyze the alias effect in mapping.

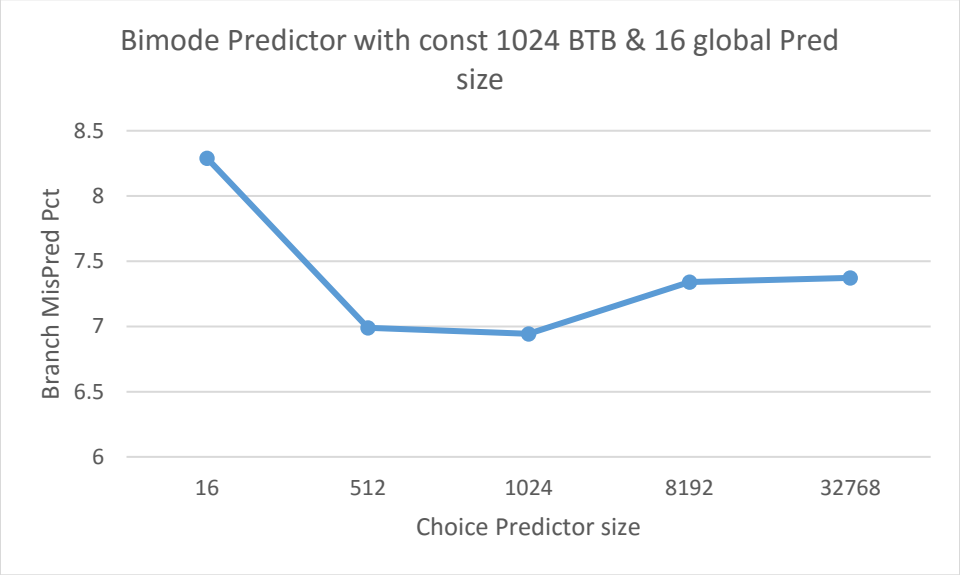


Figure 16: Graph on effect due to choice predictor size

The choice predictor size is optimum for the particular lbm benchmark at 1024. When the choice predictor size increases, small global predictor size cause aliasing.

EFFECT DUE TO GLOBAL PREDICTOR SIZE:

In the Bimode Predictor, the size of the global Predictors plays crucial role in the prediction. Here, for the current analysis, the BTB Entry size is kept constant to 1024 and the 16 choice Predictor size is kept constant and the size of the choice predictor is varied and its effect on the BranchPredictor parameters are plotted.

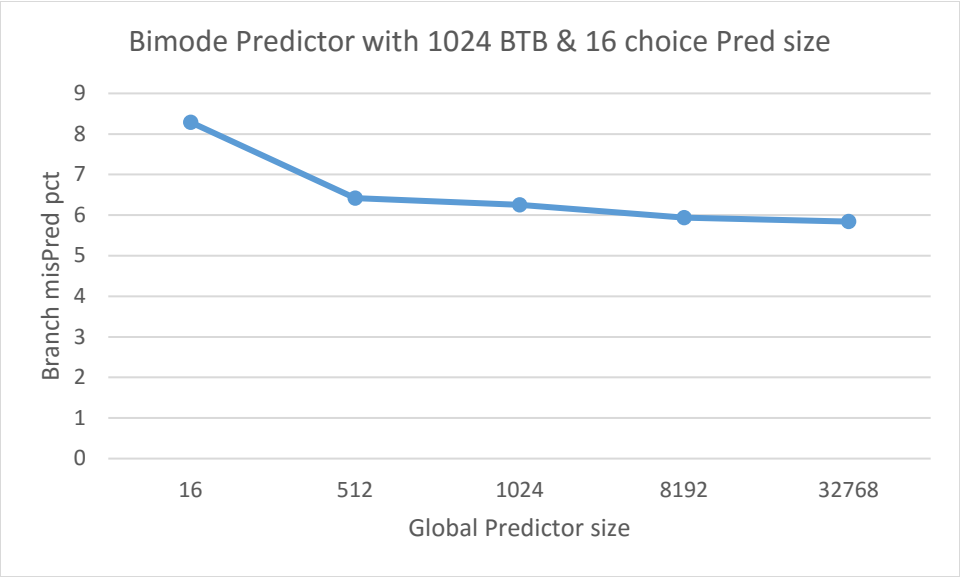


Figure 17: Graph on effect due to global predictor size

ANALYSIS OF BRANCH PREDICTORS

BIMODE PREDICTOR:

The CPI is calculated from the formula

$$CPI = 1 + \frac{((L1d_{miss_{inst.}} + L1i_{miss_{inst}})x4) + (L2_{miss_{inst}}x80)}{Instruction\ count} + \frac{BranchMispredPercent}{100} x\ BranchMispred\_Penalty$$

Here, the L1miss,L2miss almost remains constant. The BranchMispredPercent proportional to the CPI. The Branch penalty for each mispredicted branch is arbitrarily assumed to be 15 cycles.

The graph is plotted for all the combinations as shown.

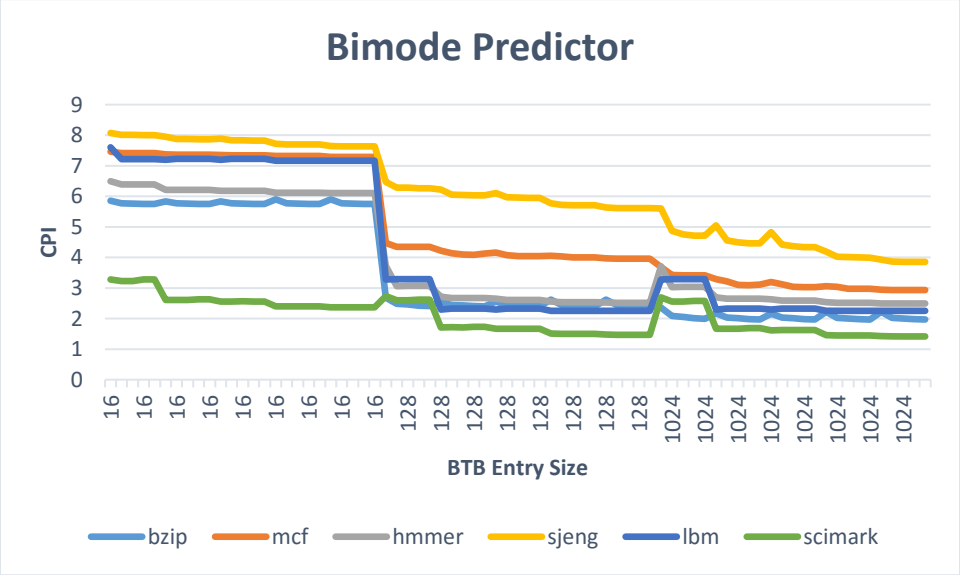


Figure 18: Graph for CPI in Bimode

The trend shows the decrease in CPI, which indicates that for large entry size mispredictions are less and we can get improved performance.

DATA ANALYSIS:

The Parameters such as total Branches, Lookups, BTBLookups, Predicted Branches are extracted from the stats.txt file and are tabulated for one case.

BTBEntries	globalPred_size	choicePred_size	lookups	BTBLookups	Branches_not_in_BTB
1024	512	512	38082430	30690890	7391540
BTBHits	**BTBHitPct**	**BTBMissPct**	**CPI**	**Branches**	**predictedBranches**
30570278	99.60701	0.39299	2.029697	38082430	31312133

BranchMispred	BranchMispredPercent	unpredictable	predictable_pct	unpredictable_pct
2311776	6.070453	6770297	82.22199	17.77801

The unpredictable branches, Efficiency, CPI are manipulated from the data obtained.

branches_not_stored_in_BTB = lookups - BTBLookups

unpredictable = Branches - predictedBranches

*predictable_pct = (predictedBranches/Branches)*100*

*unpredictable_pct = (unpredictable/Branches)*100*

*cpi = 1 + ((((l1i_miss_value+l1d_miss_value)*4)+(l2_miss_value*80))/float(inst_cnt)) + ((BranchMispredPercent/100)*15)*

LOCAL PREDICTOR:

CPI is calculated and the graph is plotted for all the combinations as shown.

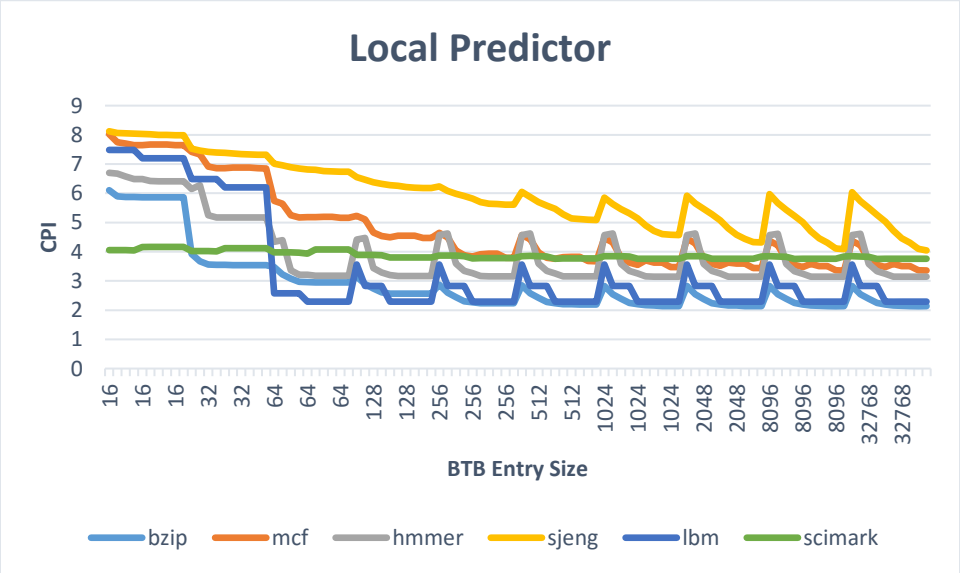


Figure 19: Graph for CPI in Local Predictor

ANALYSIS:

The overall trend decreases indicating that the mispredictions reduce with increase in the size. The spikes in the graph for the constant BTBentry size is due to 16 to 32768 Local and Choice Predictor size selections. The misprediction increases due to aliasing and improper predictor size map. (e.g. uneven sizes of BTBEntries and predictor sizes, causing aliasing)

Plotting BP params Vs CPI Vs MisPredPct

In this step, the main aim is to analyze the various cache parameters and its cost and CPI trend changes from the data obtained from running different combinations of each benchmark. Graphs are plotted against each parameter of the Cache Configuration with others parameters as constant to analyze the change in the CPI and cost-factor. These graphs are plotted using another python script that analyzes all the data from benchmark runs and extracting/filtering sub-groups which have one parameter varying and other parameters constant, which helps in plotting meaningful graphs.

Script to plot graph for analysis:

```
def plot_graphs_local(filename,testname):
    # open the file in universal line ending mode
    plot_dir = './plots/LocalBP/'+testname+'/localPred_size/'
    with open(filename, 'rU') as infile:
    # read the file as a dictionary for each row ({header : value})
        reader = csv.DictReader(infile)
        data = {}
        for row in reader:
            for header, value in row.items():
                try:
                    data[header].append(value)
                except KeyError:
                    data[header] = [value]

    # extract the variables you want
    BTBEntries_set = [float(i) for i in data['BTBEntries']]
    localPred_size_set = [float(i) for i in data['localPred_size']]
    lookups_set = [float(i) for i in data['lookups']]
    BTBLookups_set = [float(i) for i in data['BTBLookups']]
    branches_not_stored_in_BTB_set = [float(i) for i in data['Branches_not_in_BTB']]
    BTBHits_set = [float(i) for i in data['BTBHits']]
    BTBHitPct_set = [float(i) for i in data['BTBHitPct']]
    BTBMissPct_set = [float(i) for i in data['BTBMissPct']]
    Branches_set = [float(i) for i in data['Branches']]
```

```
        predictedBranches_set = [float(i) for i in data['predictedBranches']]
        BranchMispred_set = [float(i) for i in data['BranchMispred']]
        BranchMispredPerce_set = [float(i) for i in data['BranchMispredPercent']]
        unpredictable_set = [float(i) for i in data['unpredictable']]
        predictable_pct_set = [float(i) for i in data['predictable_pct']]
        unpredictable_pct_set = [float(i) for i in data['unpredictable_pct']]
        cpi_set    = [float(i) for i in data['CPI']]
        BTBEntries_uniq = list(set(BTBEntries_set))
        localPred_size_uniq = list(set(localPred_size_set))
        lookups_uniq = list(set(lookups_set))
        BTBLookups_uniq = list(set(BTBLookups_set))
#branches_not_stored_in_BTB_uniq = list(set(Branches_not_stored_in_BTB_set))
        BTBHits_uniq = list(set(BTBHits_set))
        BTBHitPct_uniq = list(set(BTBHitPct_set))
        BTBMissPct_uniq = list(set(BTBMissPct_set))
        Branches_uniq = list(set(Branches_set))
        predictedBranches_uniq = list(set(predictedBranches_set))
        BranchMispred_uniq = list(set(BranchMispred_set))
        BranchMispredPerce_uniq = list(set(BranchMispredPerce_set))
        unpredictable_uniq = list(set(unpredictable_set))
        predictable_pct_uniq = list(set(predictable_pct_set))
        unpredictable_pct_uniq = list(set(unpredictable_pct_set))
        full_list =
zip(BTBEntries_set,localPred_size_set,BTBMissPct_set,BranchMispredPerce_set,unpredictable_pct_set,cpi_set)
#####################          localpredsize          ##################################
        for BTBEntries_var in BTBEntries_uniq:
                for index in range(len(localPred_size_set)):
                        if (BTBEntries_var== BTBEntries_set[index]):
                                index_set.append(index)
                                needed_list.append((index,full_list[index]))
                if len(index_set)>=3:
                        needed_list = sorted(needed_list, key=lambda x:x[1][1])
                        sorted_index = [item[0] for item in needed_list]
                        for index in sorted_index:
                                needed_local.append(localPred_size_set[index])
```

```
            needed_BTBmiss.append(BTBMissPct_set[index])
            needed_mispred.append(BranchMispredPerce_set[index])
            needed_unpred.append(unpredictable_pct_set[index])
            needed_cpi.append(cpi_set[index])

host = host_subplot(111, axes_class=AA.Axes)
plt.subplots_adjust(right=0.75)

par1 = host.twinx()
par2 = host.twinx()

offset = 60
new_fixed_axis = par2.get_grid_helper().new_fixed_axis
par2.axis["right"] = new_fixed_axis(loc="right",
                                    axes=par2,
                                    offset=(offset, 0))

par2.axis["right"].toggle(all=True)

host.set_xlim(min(needed_local)-16, max(needed_local)+256)
#host.set_ylim(min(needed_cpi), max(needed_cpi))
#host.set_xticks(needed_local)
host.set_xlabel('localPred_size with BTBentries='+str(int(BTBEntries_var)))
host.set_ylabel("mispred_pct")
par1.set_ylabel("BTBmiss")
#par2.set_ylabel("unpred_pct")
par2.set_ylabel("CPI")

p1, = host.plot(needed_local, needed_mispred, '*-', label="mispred_pct")
p2, = par1.plot(needed_local, needed_BTBmiss, '+-',label="BTBmiss")
#p3, = par2.plot(needed_local, needed_unpred, '.-',label="unpred_pct")
p3, = par2.plot(needed_local, needed_cpi, '.-',label="CPI")

#par1.set_ylim(min(needed_wc)-200, max(needed_cost)+300)
#par2.set_ylim(min(needed_wc)-200, max(needed_cost)+300)
```

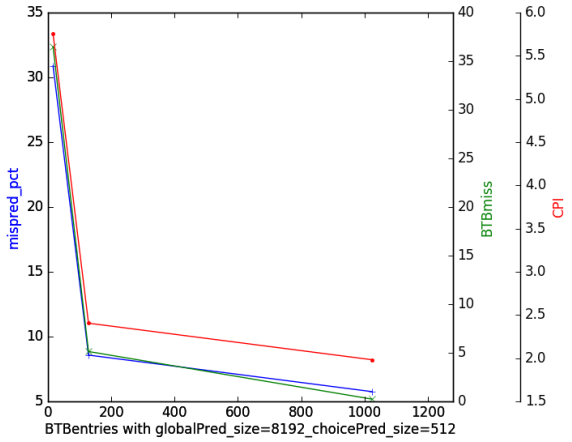
```
                    #host.legend(loc='upper left')


                    host.axis["left"].label.set_color(p1.get_color())
                    par1.axis["right"].label.set_color(p2.get_color())
                    par2.axis["right"].label.set_color(p3.get_color())
                    plt.savefig(plot_dir+testname+"_BTBEntries="+str(int(BTBEntries_var))+".png")
                    plt.close('all')
                    *********function ends**********
##########################################################################################
**********Calling fuction to execute******************************
for filename in glob.glob("./LocalBP*.csv"):
        (f_path, f_name) = os.path.split(filename)
        (f_short_name, f_extension) = os.path.splitext(f_name)
        f_short_name = f_short_name.replace("LocalBP_","",1)
        print f_short_name
        print filename
        if not os.path.exists("./plots/LocalBP/"+f_short_name):
                os.makedirs("./plots/LocalBP/"+f_short_name)
        if not os.path.exists("./plots/LocalBP/"+f_short_name+"/BTBEntries"):
                os.makedirs("./plots/LocalBP/"+f_short_name+"/BTBEntries")
        if not os.path.exists("./plots/LocalBP/"+f_short_name+"/localPred_size"):
                os.makedirs("./plots/LocalBP/"+f_short_name+"/localPred_size")
        plot_graphs_BTB(filename,f_short_name)
        plot_graphs_local(filename,f_short_name)
```

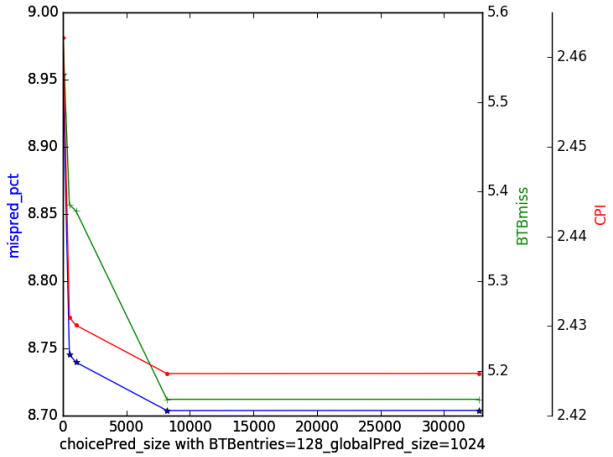
BiMode Predictor:

Bzip2 plots:

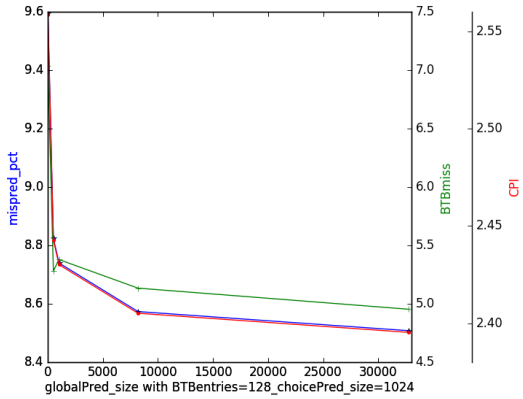
Keeping other parameters as constant, BTBEntries size, Local Predictor, Global Predictor and choice Predictor size is varied and parameters such as mispredict percent , CPI and BTBMissPct are plotted for all combinations to analyze the trend in each graph.



a)BTBEntries is varied

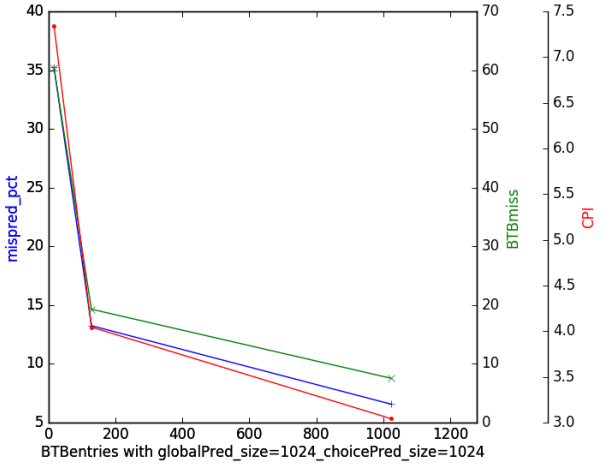


b)Choice Predictor Size is varied

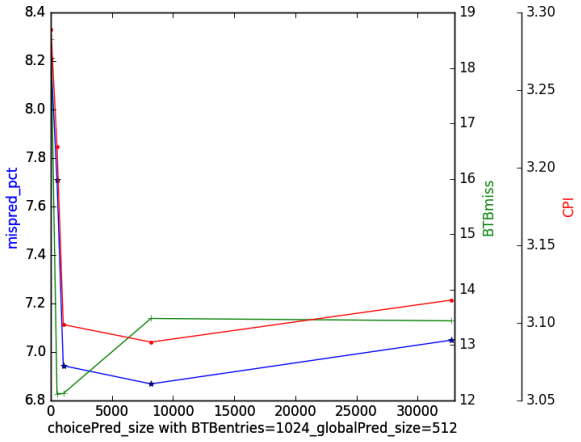


c)Global Predictor size is varied.

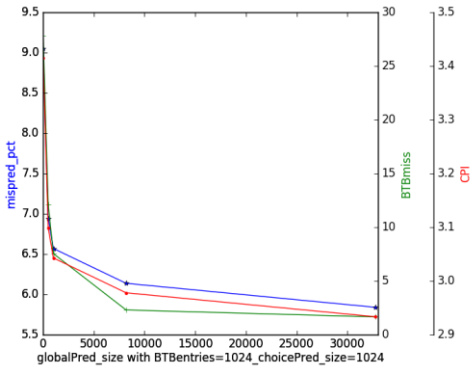
Mcf plots:



a)BTBEntries is varied

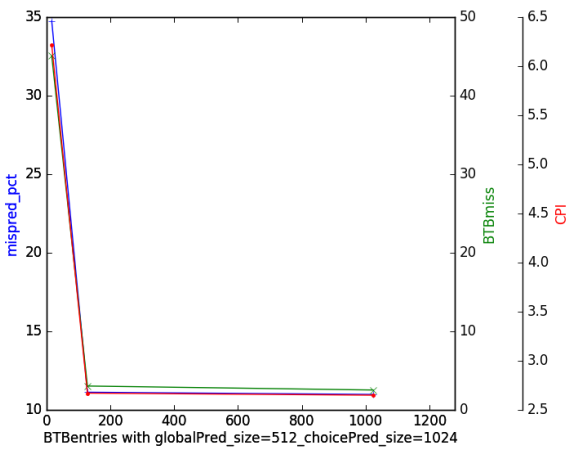


b)Choice Predictor Size is varied

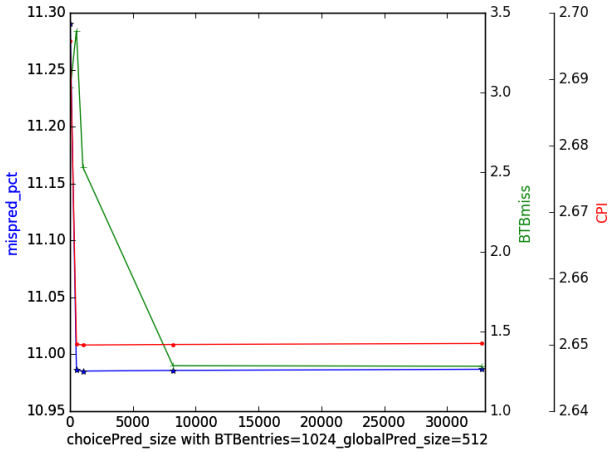


c)Global Predictor size is varied.

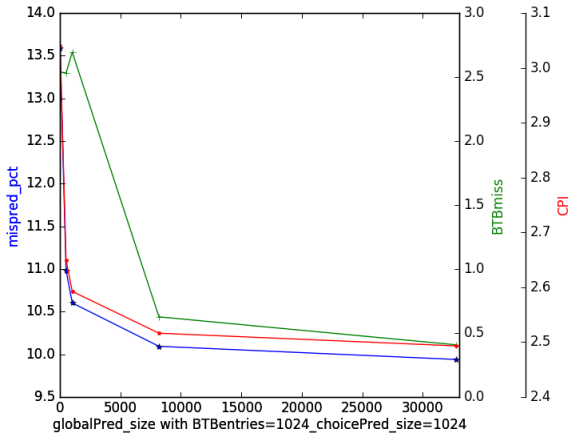
hmmer plots:



a)BTBEntries is varied

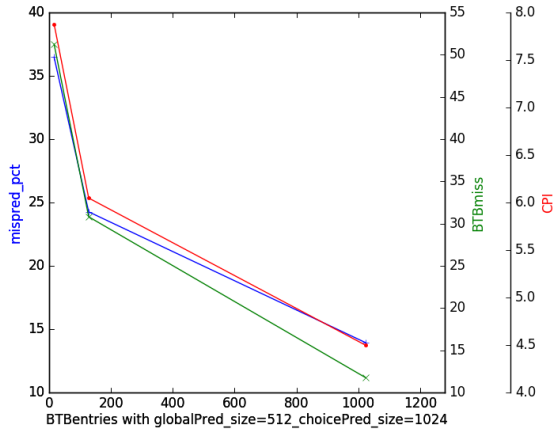


b)Choice Predictor Size is varied

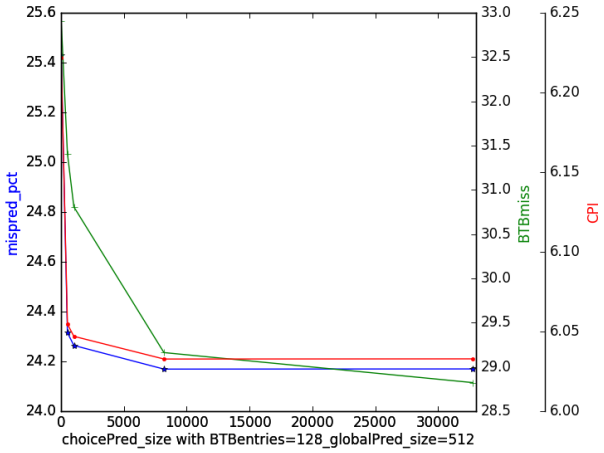


c)Global Predictor size is varied.

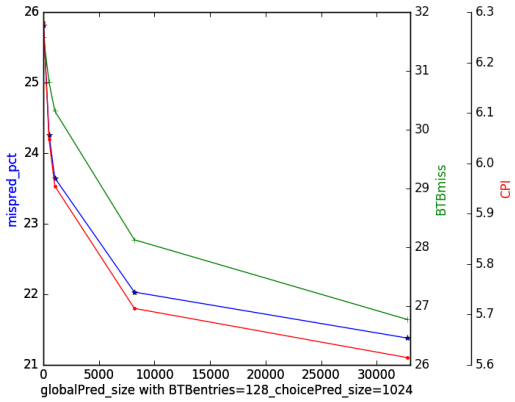
sjeng plots:



a)BTBEntries is varied

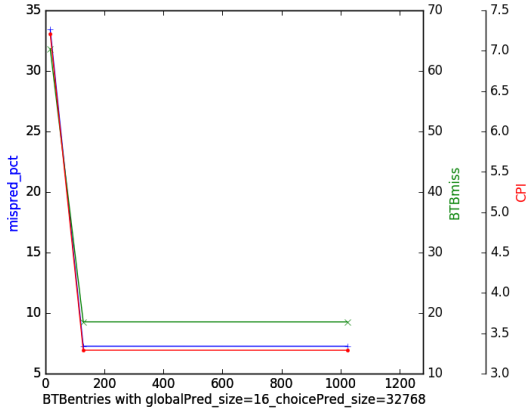


b)Choice Predictor Size is varied

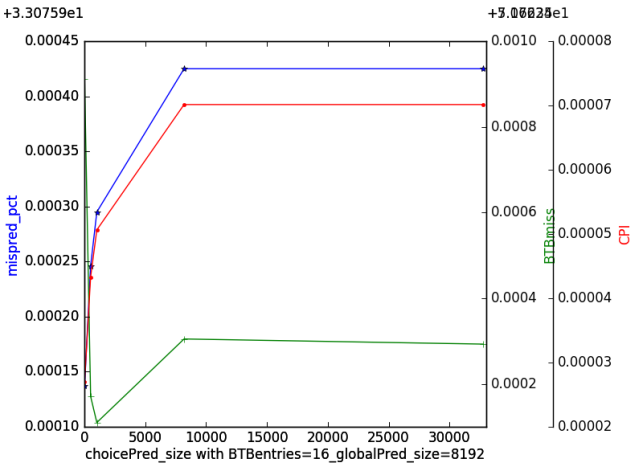


c)Global Predictor size is varied.

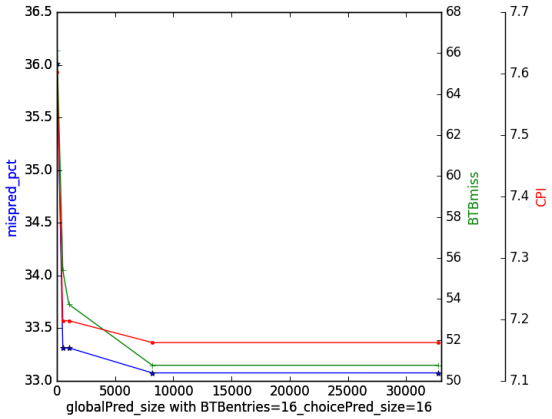
lbm plots:



a)BTBEntries is varied

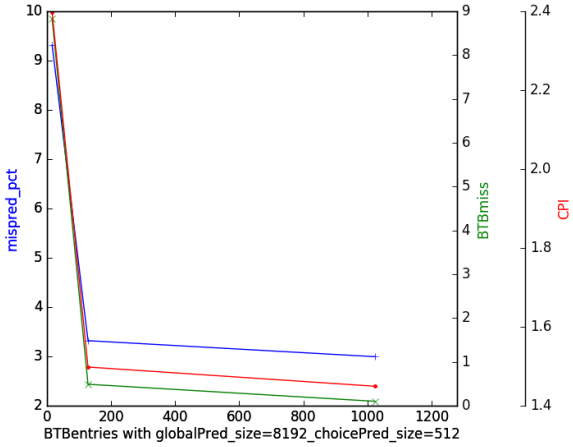


b)Choice Predictor Size is varied

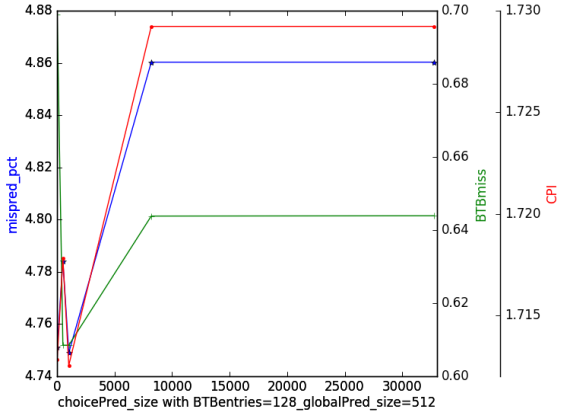


c)Global Predictor size is varied.

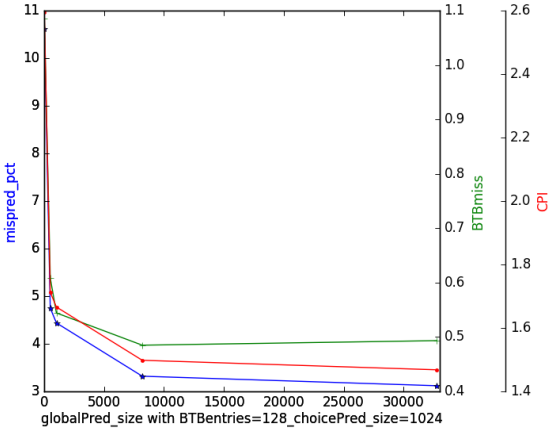
scimark plots:



a)BTBEntries is varied



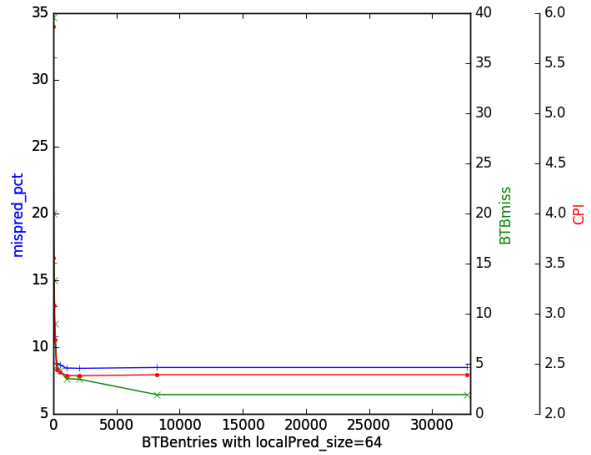
b)Choice Predictor Size is varied



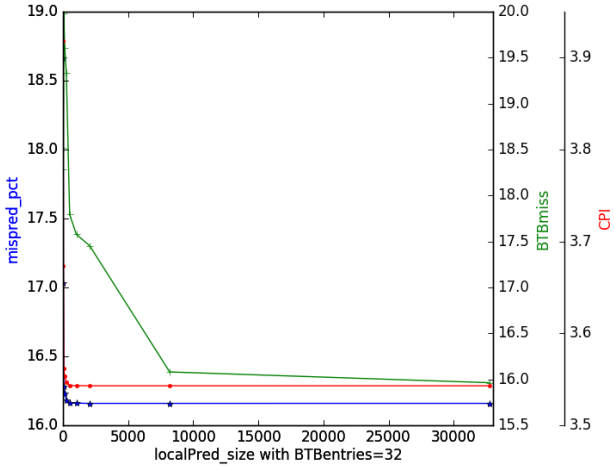
c)Global Predictor size is varied.

Local Predictor:

Bzip2 plots:

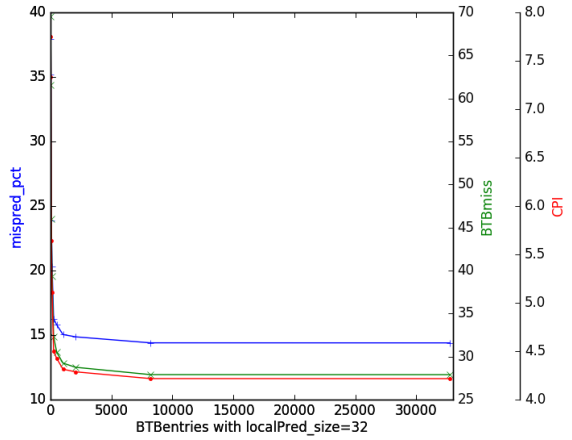


a)BTBEntries are varied

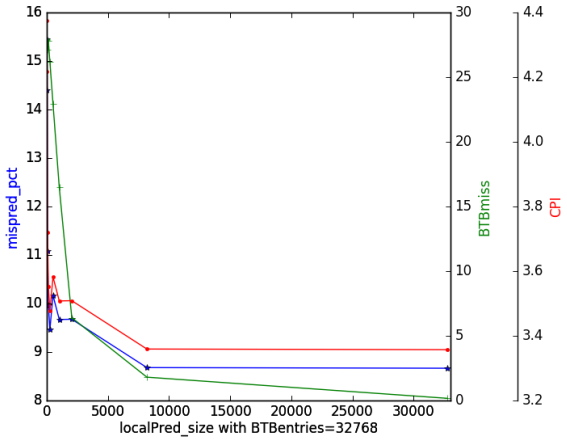


b)local predictor size is varied

Mcf plots:

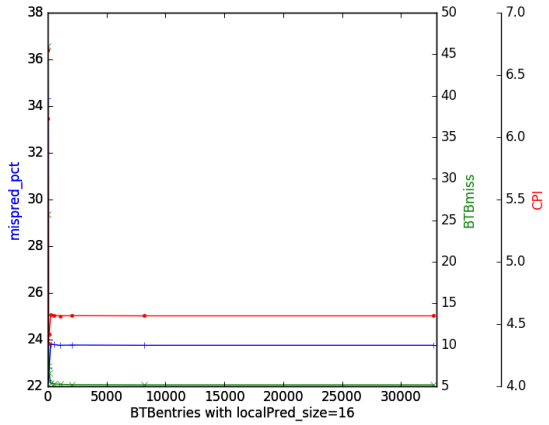


a)BTBEntries are varied

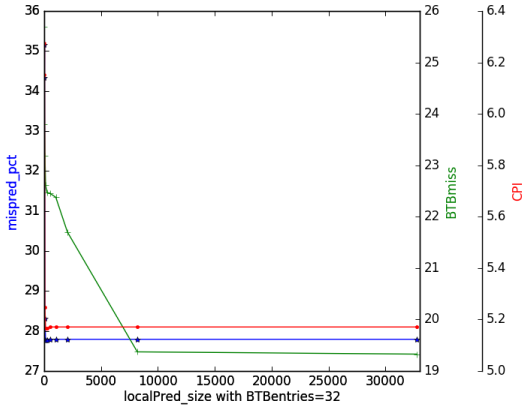


b)local predictor size is varied

hmmer plots:

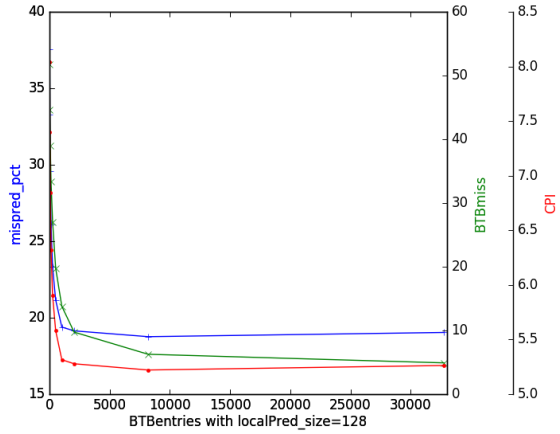


a)BTBEntries are varied

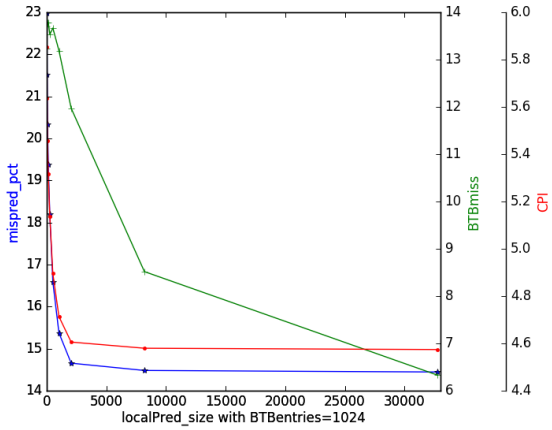


b)local predictor size is varied

sjeng plots:

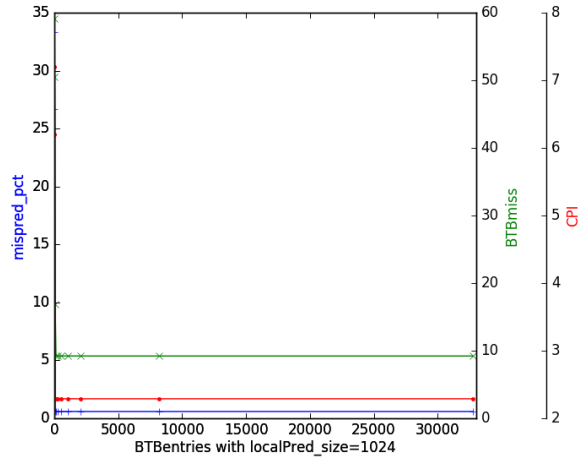


a)BTBEntries are varied

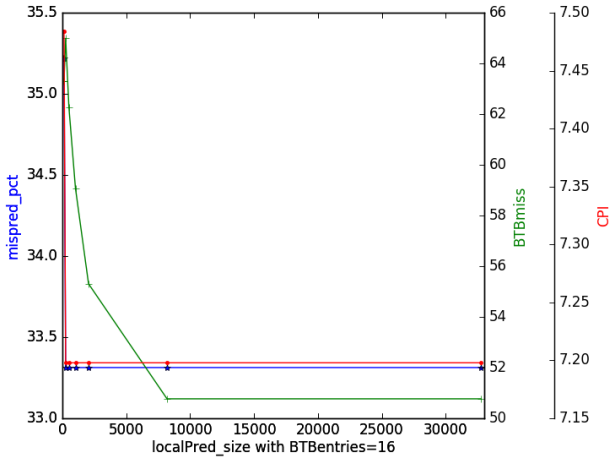


b)local predictor size is varied

lbm plots:

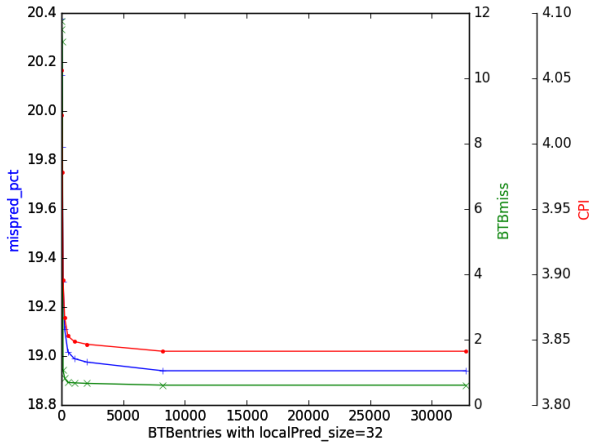


a)BTBEntries are varied

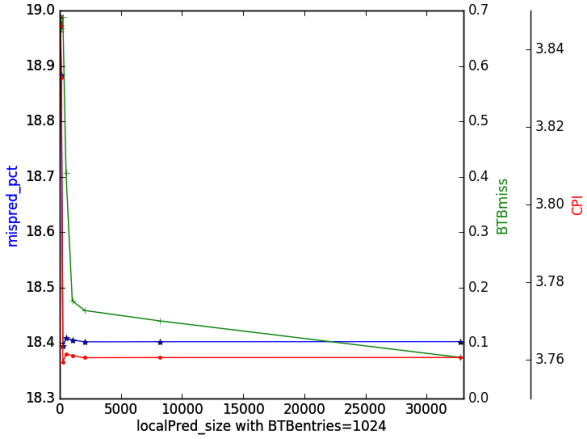


b)local predictor size is varied

scimark plots:



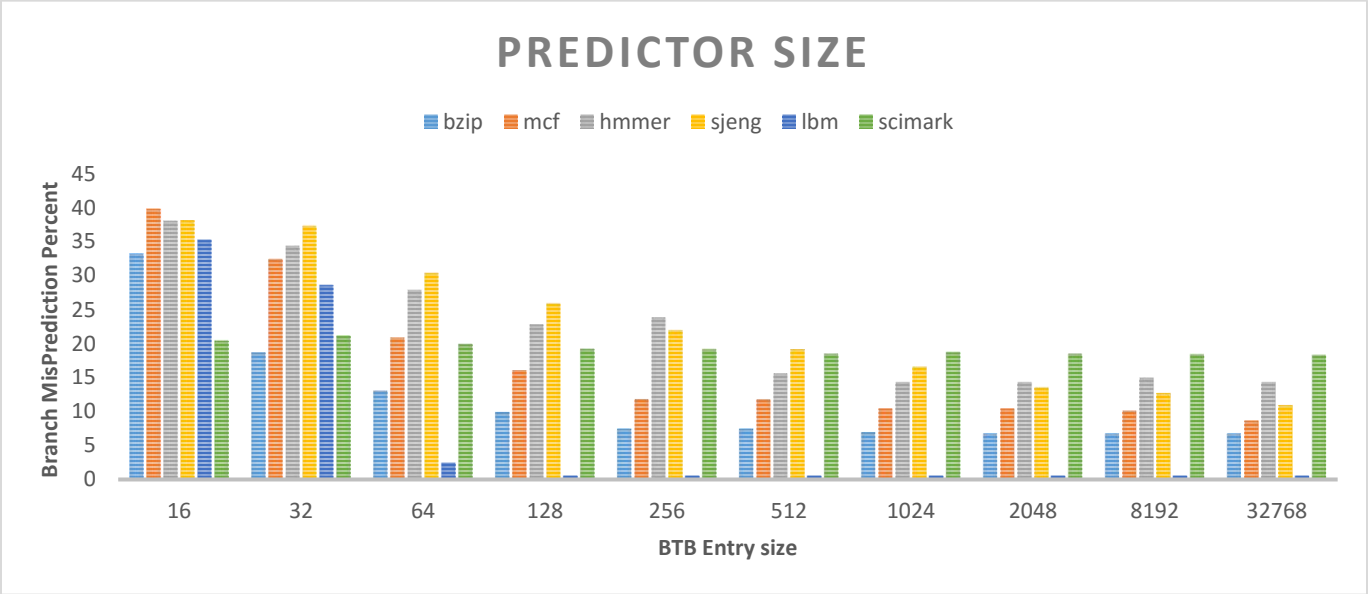
a)BTBEntries are varied



b)local predictor size is varied

PART 4: Optimize Predictor size for performance

It has been observed in the analysis that for the BTBEntry size of 1024- found to be the optimum BTBentry size with less misprediction percent. Correspondingly, Local predictor and Global Predictor size of 512 has the minimum BTBMiss Pct, thus pushing up the overall efficiency of the Branch Predictors.



ANALYSIS:

The choice of Predictor size also determines the mispredict rate. When Entry size is chosen to be 1024, the Predictor size of 256 has the minimum miss predictions. From the all the above benchmarks the following Predictor size holds good .

Local Predictor:

Optimum BTBEntry size=1024

Local Predictor size=256

Bimode Predictor:

Optimum BTBEntry size=1024

Global Predictor size=256

Choice Predictor size=256

SAMPLE COMMAND LINE:

Bimode Predictor

The sample command line for the benchmark is given below.
/home/jayaram/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/jayaram/comp_arch/Prj2/Project1_SPEC/401.bzip2/m5out/ BiModeBP_bzip2_16BTBEntries_16global_16choice /home/jayaram/comp_arch/gem5-stable/configs/example/se.py -c /home/jayaram/comp_arch/Prj2/Project1_SPEC/401.bzip2/src/benchmark -o '/home/jayaram/comp_arch/Prj2/Project1_SPEC/401.bzip2/data/input.program 10' -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=1024kB --l1d_assoc=2 --l1i_assoc=2 --l2_assoc=4 --cacheline_size=64

Local Predictor

The sample command line for the benchmark is given below.

/home/kalyan/comp_arch/gem5-stable/build/X86/gem5.opt -d /home/ kalyan /comp_arch/Prj2/Project1_SPEC/458.sjeng/m5out/ LocalBP_sjeng_16BTBEntries_16global_16choice /home/ kalyan /comp_arch/gem5-stable/configs/example/se.py -c /home/ kalyan /comp_arch/Prj2/Project1_SPEC/458.sjeng/src/benchmark -o /home/ kalyan /comp_arch/Prj2/Project1_SPEC/458.sjeng/data/test.txt -I 500000000 --cpu-type=timing --caches --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=1024kB --l1d_assoc=2 --l1i_assoc=2 --l2_assoc=4 --cacheline_size=64

CONCLUSION:

Thus, various Predictor sizes for Bimode and Local Branch Predictor is chosen for timing based CPU type and is analyzed in the gem5 simulator. Various parameters such as BTBEntry size, Local Predictor size, Global Predictor size, Choice Predictor size are varied to get the minimum BTBHitPct, MissPct, BranchMispredPercent , Accuracy and CPI for 6 individual benchmarks, namely 401.bzip2, 429.mcf, 456.hmmer, 458.sjeng, 470.lbm, scimark. At the final stage of the project, optimum Predictor size required to run the benchmarks are defined.

References:

1) *Computer Architecture A Quantitative Approach 5th Edition by John L. Hennessy and David A. Patterson*
2) https://en.wikipedia.org/wiki/Branch_predictor - *Branch Prediction - local, hybrid, global predictors reference.*
3) *www.stackoverflow.com – python script building and reference*
4) *http://gem5.org/PARSEC_benchmarks*
5) *http://www.m5sim.org/SPEC_CPU2006_benchmarks*
6) *Computer Organization and Architecture: Designing for Performance (Hardcover) by William Stallings*
7) *https://markgottscho.wordpress.com/2014/09/20/tutorial-easily-running-spec-cpu2006-benchmarks-in-the-gem5-simulator/*
8) *https://www.spec.org/cpu2006/Docs*
9) http://math.nist.gov/scimark2/
10) *Gem5 src : src/cpu/pred/bi_mode.hh*
11) *Gem5 src : src/cpu/pred/2bit_local.hh*
12) *Gem5 src : src/cpu/pred/tournament.hh*